Lexical Analysis, I

Comp 412
Our goal is to automate the construction of scanners & parsers

**Scanner**
- Specify syntax with regular expressions (REs)
- Construct finite-automaton & scanner from the RE

**Parser**
- Specify syntax with context-free grammars (CFGs)
- Construct push-down automaton & parser from the CFG
In Lecture 2, we saw some ambiguity in defining “positive integer”

- Is 001 a positive integer? What about 00?
- The automata are precise specifications, but the words are not

---

**Big Picture**

We need a better notation for specifying microsyntax than these transition diagrams.
Regular Expressions

**We need a better notation for specifying microsyntax**

*“better” ⇒ both formal and constructive*

**Regular Expressions over an Alphabet \( \Sigma \)**

- If \( x \in \Sigma \), then \( x \) is an RE denoting the set \( \{ x \} \) or the language \( L = \{ x \} \)
- If \( x \) and \( y \) are REs then
  - \( xy \) is an RE denoting \( L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \} \)
  - \( x \mid y \) is an RE denoting \( L(x) \cup L(y) \)
  - \( x^* \) is an RE denoting \( L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k \) (**Kleene Closure**)
    - Set of all strings that are zero or more concatenations of \( x \)
  - \( x^+ \) is an RE denoting \( L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k \) (**Positive Closure**)
    - Set of all strings that are one or more concatenations of \( x \) (or \( xx^* \))
- \( \varepsilon \) is an RE denoting the empty set

Many RE-based systems support additional notation and operators. Those added features build on alternation, concatenation, and closure — plus, perhaps logical complement or negation. Complement is easy and efficient, if we think of the underlying DFA. (We will revisit this issue.)
Regular Expressions

How do these operators help?

Regular Expressions over an Alphabet $\Sigma$

- If $x$ is in $\Sigma$, then $x$ is an RE denoting the set $\{x\}$ or the language $L = \{x\}$
  $\rightarrow$ *The spelling of any letter in the alphabet is an RE*

- If $x$ and $y$ are REs then
  - $xy$ is an RE denoting $L(x)L(y) = \{pq \mid p \in L(x) \text{ and } q \in L(y)\}$
    $\rightarrow$ *If we concatenate letters, the result is an RE, so we can spell words*
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
    $\rightarrow$ *Any finite list of words can be written as an RE, $(w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n)$*
  - $x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$
  - $x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$
    $\rightarrow$ *We can use closure to write finite descriptions of infinite, but countable, sets*

- $\epsilon$ is an RE denoting the empty set
  $\rightarrow$ *$\epsilon$ is sometimes useful for writing more concise REs*
Regular Expressions

Let the notation \([a...z]\) be shorthand for the RE

\[(a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z)\]

Examples

Tasteless positive integer \([0...9] [0...9]^*\)

or \([0...9]^*\)

Tasteful positive integer \(0 | [1...9] [0...9]^*\)

Identifier (Algol-like lang) \(([a...z] | [A...Z]) ([a...z] | [A...Z] | [0...9])^*\)

Decimal number \(0 | [1...9] [0...9]^* . [0...9]^*\)

Real number \(((0 | [1...9] [0...9]^*)) | (0 | [1...9] [0...9]^* . [0...9]^*) E [0...9] [0...9]^*)\)

Each of these REs corresponds to an automaton. More precisely, they correspond to a **deterministic finite automaton, or DFA.**

- **Deterministic:** at each point, it makes a consistent, predictable decision
- **Finite:** a bounded number of states in the automaton
Why do we care about regular expressions in the context of a compiler?

- We use **REs** to specify *microsyntax* —- the mapping of spelling to parts of speech
  - An identifier is ( [a...z] | [A...Z] ) ( [a...z] | [A...Z] | [0...9] )*
  - Keywords are specified by their spellings, e.g., *if, then, else*
  - Those spellings are, in turn, **REs**

- We use tools derived from automata theory to derive a **DFA** from the **REs** and then convert the **RE** to code that implements a scanner
  - Automatic construction reduces the time & cost of scanner construction
  - Derivation from a formal notation eliminates implementation errors
  - Resulting scanners are both **efficient** ($O(n)$) and **fast** (*low constant overhead*)

- **RE**-derived scanners are widely used
  - Compilers, text editors, input checking on web pages, software to filter **URLs**
Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. ...  

Regular expression patterns are **compiled into a series of bytecodes which are then executed by a matching engine** written in C. For advanced use, **it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization** isn’t covered in this document, because it **requires that you have a good understanding of the matching engine’s internals.**  

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

*From Python 2.7.10 documentation, emphasis added*

- You will learn how to “compile” REs to a **DFA** & implement a **DFA**  
  - Execution cost is guaranteed $O(1)$ per input character, *independent* of the expression
- You will have deeper understanding of their power & their use
A Digression on Time

In COMP 412, we will talk about a lot of “times”

• Design time, build time, compile time, run time, ...

• In practice, the issue of when something happens is one that causes a great deal of confusion among students of compiler construction
  ➜ Costs incurred at design or implementation time do not increase compile time

• Compile time happens every time the user invokes the compiler
  ➜ Users are, appropriately, sensitive to compile time
  ➜ Costs incurred at compile time do not increase run time

• Run-time costs affect actual application performance
  ➜ One critical goal for compilation is to keep run time to a minimum, which means reducing the overhead introduced by translation

As we look at strategies for generating scanners & parsers, keep in mind that generation costs are incurred at implementation time
Automatic Scanner Construction: Meta Issues

**Goals**

- Simplify the construction of robust, efficient scanners
- Develop techniques that have widespread applicability
- Understand the underlying theory & practice

---

1. We write REs at design time

2. Tools generate the scanner at build time
   - **Scanner Generator**
     - specifications written as regular expressions
     - e.g., lex, flex

3. When the compiler runs, it uses the generated scanner to convert source code into a stream of tokens.

---

**Diagram**

- **Source code**
- **Scanner**
- **Stream of <word, category> pairs**
- **Compile time**
- **Design & build times**

---

**Notes**

- Tools generate the scanner at build time
- We write REs at design time
- When the compiler runs, it uses the generated scanner to convert source code into a stream of tokens.
Automatic Scanner Construction

Scanner Generator

• May encode its knowledge in tables that drive a “skeleton scanner”
  – Skeleton scanner interprets the tables to simulate the DFA
  See § 2.5.1
• Every scanner uses the same skeleton, independent of RE
• Scanner generator builds the DFA from the RE, & converts it to a table
Automatic Scanner Construction

Scanner Generator

- May encode its knowledge of the recognizer directly into code
  - Transitions are compiled into conditional logic
- Scanners for different REs are different
- Produces a scanner that has very low overhead per character
- Scanner generator builds the DFA from the RE, & emits code for it

See § 2.5.2
Example from Lecture 2

Recognizer for an ILOC register name (allow redundant zeros)

Rules for DFA Operation

- Start in state $s_0$ & make transitions on each input character
- **DFA** accepts a word $x$ if and only if $x$ leaves the **DFA** in an accepting or final state
- If the **DFA** encounters a character with no specified transition, it moves to $s_e$ & stays in that state
- $r17$ takes it through $s_0, s_1, s_2, s_2$ and it accepts
- $r$ takes it through $s_0, s_1$ and it fails
- $ra$ takes it through $s_0, s_1$, and $s_e$, so it fails

We will use the RE for a register name as a continuing example.
Example

To be useful, the DFA must be executable

```
char ← next character
state ← s₀
while (char ≠ EOF) {
    state ← δ[state, char]
    char ← next character
}
if (state is a final state)
    then report success
else  report failure
```

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Skeleton Scanner

Transition Table (δ)

This skeleton scanner is simplified. See Figure 2.14 in § 2.5.1 of EaC2e and the accompanying text.

O(1) per character

Character classifier maps any character into one of the 3 classes: {r}, {0...9}, {all others}
Example

To capture and classify the lexeme, we add a little work to each state

char $\leftarrow$ next character
state $\leftarrow$ $s_0$
lexeme $\leftarrow$ null string
while (char $\neq$ EOF) {
  lexeme $\leftarrow$ lexeme | | char
  state $\leftarrow$ $\delta$[state,char]
  char $\leftarrow$ next character
}

If (state is a final state) then {
  category $\leftarrow$ $f$[state]
  return <lexeme,category>
}
else report failure

Transition Table ($\delta$)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4</th>
<th>5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
</tbody>
</table>

Skeleton Scanner

Still O(1)
Example

To capture the register number, we would need state-specific actions

char $\leftarrow$ next character
state $\leftarrow$ $s_0$
while (char $\neq$ EOF) {
    state $\leftarrow$ $\delta$[state,char]
    char $\leftarrow$ next character
    if (state = $s_1$)
        n $\leftarrow$ 0
    else if (state = $s_2$)
        n $\leftarrow$ n * 10 + char – ‘0’
} 

If (state is a final state) then {
    category $\leftarrow$ $f$(state)
    return <lexeme,category>
} else report failure

Transition Table ($\delta$)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Still $O(1)$
More Complex REs

What about a more complex language?
• $r [0...9] [0...9]^* \text{ allows arbitrary register numbers (e.g., } r000 \text{ or } r999)$
• What if we want to limit the register name to $r0$ through $r31$?

Write a tighter specification into the RE
• $r ( (0|1|2) ([0...9] | \varepsilon) | (4|5|6|7|8|9) | (3|30|31) )$
• $r0|r1|r2|r3| \ldots |r31|r00|r01|r02| \ldots |r09$

Each of these REs can be converted to a DFA
• The DFA has the same $O(1)$ cost per transition
• The DFA takes one transition per input character
• The DFA uses the same skeleton scanner

The added complexity is in the RE, not in the scanner†

† recall the Python documentation
More Complex REs

The DFA for $r ( (0|1|2) ([0...9] | \epsilon) | (4|5|6|7|8|9) | (3|30|31) )$

- Accepts a more constrained set of register names
- Still $O(1)$ cost per input character
- More states $\Rightarrow$ more rows in the transition table $\Rightarrow$ more memory
More Complex REs

The DFA for $r \ (0|1|2) \ ([0...9] \ | \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) \ )$

- Accepts a more constrained set of register names
- Still $O(1)$ cost per input character
- More states $\Rightarrow$ more rows in the transition table $\Rightarrow$ more memory

Automata Theory Moment
Earlier, we said we would revisit logical complement of an RE or a DFA.
To complement a DFA:
- Make non-final states into final states
- Make final states into non-final states
DFA then accepts any string that the original did not accept $\Rightarrow$ its complement
This result is not obvious when thinking about the RE.
More Complex REs

The DFA for \( r \ (0|1|2) ([0...9] | \varepsilon) \ (4|5|6|7|8|9) \ (3|30|31) \) 

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>( r )</th>
<th>0, 1</th>
<th>2</th>
<th>3</th>
<th>4...9</th>
<th>Any Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_2 )</td>
<td>( s_2 )</td>
<td>( s_5 )</td>
<td>( s_4 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_e )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_3, s_4 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_e )</td>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
</tbody>
</table>

Notice that the character classifier has many more divisions that did the earlier one. Still, it should be implementable as a function with \( O(1) \) cost. (see § 2.5)

Compressed 2 states, as well

This table runs in the same skeleton scanner without changes

- To change the language, just change the table
- Still \( O(1) \) cost per character
Terminology Matters

So far, we have only looked at deterministic automata, or DFAs

- **DFA** ≡ Deterministic Finite Automaton
- Deterministic means that it has only one transition out of a state on a given character

\[ s_0 \xrightarrow{a} s_1 \]  

rather than

\[ s_0 \xrightarrow{a} s_1; s_0 \xrightarrow{a} s_2 \]
Determinism (or not)

So far, we have only looked at **deterministic automata**, or **DFAs**

- **DFA** ≡ **Deterministic Finite Automaton**
- Deterministic means that it has only one transition out of a state on a given character

![Diagram](image)

- Can a finite automaton have multiple transitions out of a single state on the same character?
  - Yes, we call such an FA a **Nondeterministic Finite Automaton**
  - And, yes, the **NFA** is one of the more odd notions in **CS** ... but a useful one

- **NFAs** and **DFAs** are equivalent
  - Sometimes, it is easier to build an **NFA** than to build a **DFA**

\[ \varepsilon \text{-transition does not consume an input character, which should worry us.} \ (O(1) \ ?) \]
Where are we going?

We will show how to construct, for any RE $r$, a deterministic finite-state automaton that recognizes $r$

Overview:

1. Simple and direct construction of a **nondeterministic finite automaton (NFA)** to recognize a given RE
   - Easy to build in an algorithmic way
   - Requires transitions on $\varepsilon$ to combine regular subexpressions

2. Construct a **deterministic finite automaton (DFA)** that simulates the NFA
   - Use a set-of-states construction

3. Minimize the number of states in the DFA
   - We will look at 2 different algorithms: Brzozowski & Hopcroft

4. Generate the scanner code
   - Additional specifications needed for the actions
The Plan for Scanner Construction

**RE → NFA** (*Thompson’s construction*)
- Build a *nondeterministic finite automaton (NFA)* for each term in the RE
- Combine them in patterns that model the operators

**NFA → DFA** (*Subset construction*)
- Build a DFA that simulates the NFA

**DFA → Minimal DFA**
- Brzozowski’s algorithm
- Hopcroft’s algorithm

**DFA → RE**
- All pairs, all paths problem
- Union together paths from \( s_0 \) to a final state

**The Cycle of Constructions**

*STOP*

**Taken together, these constructions prove that DFAs, NFAs and REs are equivalent.**
Example

What about a DFA for \((a \mid b)^* \text{abb}\)?

Each RE corresponds to one or more deterministic finite automaton (DFAs)

- We know a DFA exists for each RE
- The DFA may be hard to build directly
- Automatic techniques will build it for us...

This DFA is not particularly obvious from the RE.
Example as an **NFA**

Here is a simpler FA for \((a | b)^* \text{abb}\) — an NFA

\[\begin{array}{c}
S_0 \xrightarrow{\epsilon} S_1 \xrightarrow{a} S_2 \xrightarrow{b} S_2 \xrightarrow{b} S_3
\end{array}\]

\((a | b)^* \text{abb}\)

**Here is an NFA for the same language**
- The relationship between the **RE** and the **NFA** is more obvious
- The \(\epsilon\)-transition pastes together two **DFAs** to form a single **NFA**
- We can rewrite this **NFA** to eliminate the \(\epsilon\)-transition
  - \(\epsilon\)-transitions are an odd and convenient quirk of **NFAs**
  - Eliminating this one makes it obvious that it has 2 transitions on \(a\) from \(S_0\)
Non-deterministic Finite Automata

An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$, ignoring $\varepsilon$’s

- Transitions on $\varepsilon$ consume no input
- Two models for NFA execution
  1. To “run” the NFA, start in $s_0$ and guess the right transition at each step
  2. To “run” the NFA, start in $s_0$ and, at each non-deterministic choice, clone the NFA to pursue all possible paths. If any of the clones succeeds, accept

Why study NFAs?

- They are an interesting and powerful abstraction
- They are the key to automating the RE→DFA construction

† See page 44 in EaC2e.
Relationship between NFAs and DFAs

DFA is a special case of an NFA
• DFA has no ε transitions
• DFA’s transition function is single-valued
• Same rules will work

DFA can be simulated with an NFA
  – Obviously

NFA can be simulated with a DFA
• Simulate sets of possible states
• Possible exponential blowup in the state space
• Still, one state per character in the input stream

⇒ NFA & DFA are equivalent in ability to recognize languages

Rabin & Scott, 1959